The Ultimate Implementations For Sharping SAT

Juan Manuel Dato Ruiz
jumadaru@gmail.com

Abstract
There are various mechanisms to determine what are the solutions of a formula of Boolean algebra. But the operation which makes obsolete all others is which gives us the number of solutions that satisfy the formula. That is, how many independent sequences of assignments on Boolean variables get to satisfy the proposal. This document, paradoxically, in combination with other filings, could even invite for counting the number of solutions of a formula within the $\mathbf{Z_n}$. Always with the intention of achieving O(**n**) with **n** the size of the input, or at less getting to a polynomial bound .

It is not a bug, it is a feature.
How do you calculate the i-th number of Fibonacci in a Turing Machine? There are two easy philosophies when your entry are the digits of a number in a specific base. The easiest one is described by a loop and two temporal variables which store two numbers in each step. This way requires an exponential number of steps, because the number of steps is proportional with the number put in the entry.

But there is other philosophy which gives us a result faster: if we corroborate the ratio between the biggest consecutive numbers of Fibonacci is the Golden Ratio, then the Fibonacci expression can be written in a way

$$F_n \sim K_1 \times \Phi^n$$

Even more, considering the sucession is strictly increasing then the error will be expressed in the same way with other real number lower in absolute value.

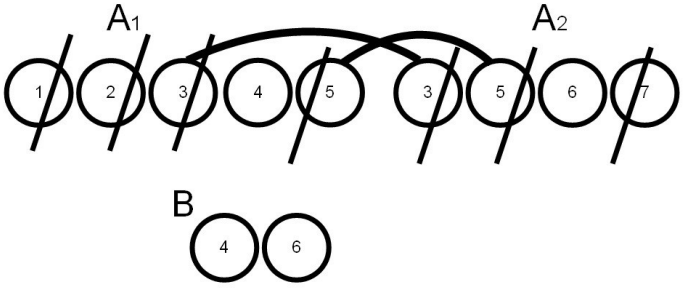
If we decide to use this other philosophy, the numbers of steps will be constant, but the result will depend of the number of digits we required for the parameters of the algorithm (i.e. $\mathbf{K_1}$ and $\mathbf{\Phi}$). So if the precision of the real numbers were not too good, then the result in great values could be inexact.

That was the real plot of this document: a good example which showed us that if our solutions were based in something similar to Fibonacci numbers, then we would get two results. One fast and appropiated in Formal Computing and one well defined and exact in Constructive Computing.

Preparing the entry
We will begin from the study of Boolean algebra converted into the *product of choices* format exposed in reference [2]. More specifically, if sets $\mathbf{A_k}$ and **B** were defined by (1), how many different **B** can be set for every list of sets $\mathbf{A_k}$.

$$\begin{gathered}\forall k \in \mathbb{N}\, A_k \in 2^{\mathbb{N}} \exists B \in 2^{\mathbb{N}}\ donde: \\ 1.\ \forall x \exists k\, x \in B \rightarrow x \in A_k \\ 2.\ \forall x \forall y \forall k\, x \in A_k \wedge y \in A_k \wedge x \in B \wedge y \in B \rightarrow x = y\end{gathered} \tag{1}$$

Under this format is easy to find equivalence formulas of logic, specifically, in (2) we see a way to associate $\mathbf{A_k}$ defined in (1) with the digital component *full adder*. For example, in (2) the element 2 in sets A_1 and A_2, can be in B if and only if proposition "p and q" is true. That assert will work simultaneously with the idea that proposition "p" is true if and only if node 1 is not in B. So, knowing how many sets B are possible give us the possibility of knowing how many cases assert a boolean formula to be true.



$$\forall k \in \mathbb{N}\, A_k \in 2^{\mathbb{N}} \exists B \in 2^{\mathbb{N}}\, where:$$
$$1.\ \forall x \exists k\ x \in B \rightarrow x \in A_k$$
$$2.\ \forall x \forall y \forall k\ x \in A_k \wedge y \in A_k \wedge x \in B \wedge y \in B \rightarrow x = y$$

This is better explained in the beggining of reference **§1**, more graphically.

$$\begin{gathered} A_1 = \{1,2,3\} \\ A_2 = \{4,2,5\} \\ A_3 = \{3,5,6\} \\ p \sim 1 \notin B \\ q \sim 4 \notin B \\ p \wedge q \sim 2 \in B \\ p \oplus q \sim 6 \notin B \end{gathered} \tag{2}$$

If we create equivalences of formulas in $\mathbf{Z_n}$ to the format of (1) for any **n**, then it will be able of managing for the modular arithmetic all properties showed in this document. We could not include every theorem of modular arithmetic, with quantifiers symbols like "for all" and "exists", if we wanted to keep the polynomial bound easily. But the objective of this document is not modeling cases of the form (1), but else we will transform it to another more suitable format for the calculating of the number of cases. To do this, we convert arrays $\mathbf{A_k}$ in ordered tuplas of three: $\mathbf{T_k}$. Ie a Tk is three ordered naturals where it is met (3). Equivalence between (1) to (3) is in reference **§1**.

$$T_k \sim \left(T_k[0], T_k[1], T_k[2]\right) \tag{3}$$

Once generatid tuples as in (3), so we want to format them in a chained $\mathbf{E_k}$ according to (4), without losing the original variability, and thereby excluding or including any case.

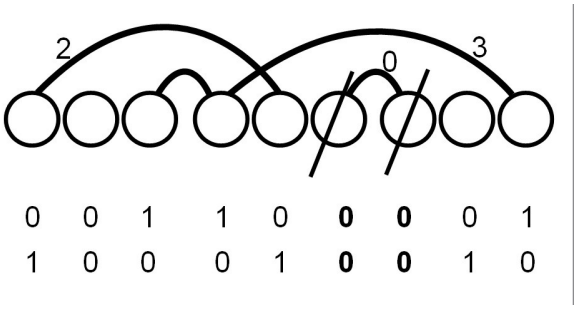
$$\forall k\, E_k[2] = E_{k+1}[0] \tag{4}$$

For processing from (3) to (4), just sort $\mathbf{T_{k'}}$ so that, for each k' which meets in $\mathbf{T_{k'}}$ property (4), we say $\mathbf{E_{k'}} = \mathbf{T_{k'}}$. If, on the contrary, component 2 does not equal 0 succesor component, then transformation (5) is applied.

$$\begin{gathered} \forall k\, T_k[2] \neq T_{k+1}[0] \rightarrow (T_k, T_{k+1}) \sim (T_k, E_1, E_2, E_3, T_{k+1}) \\ E_2[1] \in B \end{gathered} \tag{5}$$

Thanks to this approach, chained tuples conform to an easier format. We can construct a succession of nodes in the way: $\mathbf{E_0[0], E_0[1], E_1[0], E_1[1], E_2[0]...}$ The positions of every node of the tuples can be put in order where nodes in $\mathbf{E_k}[N]$ positions will be odd if and only if N is odd.

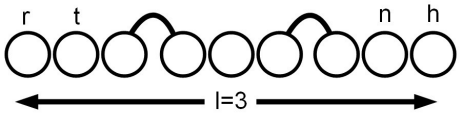
Except at the beggining and the end, the even represent what I will call *bridge,* and if we find the same node in two different tuples (and that node is not the even between two consecutives) I will call it *long bridge*. That long bridge will be represented by an unique number greater than 1, this number is called its *descriptor*.



In this way, we will get the necessary transformation: tuples in (5) are transformed in the tuple of three (**P**, **S**, **D**), where **P** is set of pairs **N x N** which contains all the correspondences between a position and its descriptor desbribed before, or more rigorously like in (6). **S** is the number of chained tuples and **D** is the number of different descriptors.

$$\begin{gathered}\forall k \geqslant 0\, \forall i \in \{0,1,2\}\, E_k[i] \sim (2 \times k + i,\, descriptor(E_k[i])) \\ descriptor(X) = \left\{ \begin{array}{c} 1 \leftarrow X \in B \\ 0 \leftarrow X \notin B \\ N \leftarrow \exists k\, \exists l\, k \neq l\, X \in E_k \wedge X \in E_l \end{array} \right\} \\ \forall X\, \forall Y\, descriptor(X) = descriptor(Y) > 1 \rightarrow X = Y \end{gathered} \tag{6}$$

From the proposed format of the tuple (**P**, **S**, **D**), now we are prepared to generate the list of tuples of five which I call *Scales*. The Scale format is close enough to get the number of cases which solves #SAT. So the next step is to transform from (**P**, **S**, **D**) to Scales. One Scale is composed by five naturals (**length**, **rattle**, **tail**, **neck**, **head**). A good interpretation of this is that is more likely to think in that easier like an extension of a list, where we add a *rattle* and a *neck*.



Length says how many chained tuples are contained in the Scale. Rattle, and tail are the values (the nodes) of the lowest tuple (if h is the lowest, $\mathbf{E_h[0]}$ for the rattle and $\mathbf{E_h[1]}$ for the tail). When neck and head are the last values of the greatest tuple ($\mathbf{E_{h+l-1}[1]}$ and $\mathbf{E_{h+l-1}[2]}$). The list of Scales represent all the information constructed from (1).

In listing 1 we can find an algorithm which transform from the format of descriptors in (6) to Scales. If we study complexity of algorithm, it is O(**S**), considering **S** proportional to the number of $\mathbf{A_k}$ in (1).

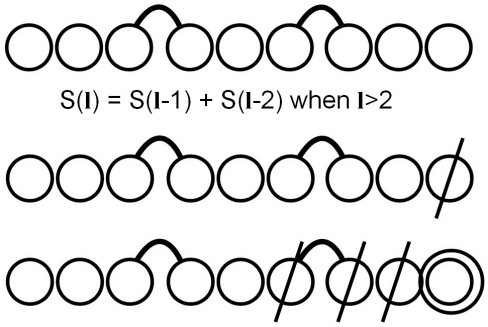
Input: pairs = A list of pairs (position, value) ordered from the first element;
lastPair = the last even position possible for pairs list.
Output: The equivalent list of Scale objects.

```
1.  if first element in pairs is not (0, ?) then
2.      add from the beggining of the list pairs: (0, None)
3.  if last element in pairs is not (0, ?) then
4.      add from the end of the list pairs: (lastPair, None)
5.  Result := empty list
6.  pairIni := 0
7.  pairEnd:= None
8.  r, t, n, h := None, None, None, None
9.  for each (pos, val) ϵ pairs (ascending forward) do
10.     if pairEnd is None then
11.         if pos is even and pos > pairIni then
12.             Append to Result: ((pos-pairIni)/2, r, t, n, val)
13.             r, t, n, h := val, None, None, None
14.             pairIni := pos
15.             pairEnd := None
16.         elif pos is even then
17.             r := val
18.         elif pos > pairIni + 1 then
19.             n := val
20.             pairEnd := pos + 1
21.         else
22.             t := val
23.     else
24.         if pos = pairEnd then
25.             h := val
26.             Append to Result: ((pairEnd – pairIni)/2, r, t, n, h)
27.             r, t, n, h := h, None, None, None
28.             pairIni := pos
29.             pairEnd := None
30.         else
31.             Append to Result: (pairEnd – pairIni)/2, r, t, n, h)
32.             if pos is even then
33.                 Append to Result: ((pos – pairIni)/2, h, None, None, val)
34.                 r, t, n, h := val, None, None, None
35.                 pairIni := pos
36.                 pairEnd := None
37.             elif pos = pairEnd +1 then
38.                 pairIni := pairEnd
39.                 pairEnd := None
40.                 r, t, n, h := h, val, None, None
41.             else
42.                 pairIni:= pairEnd
43.                 pairEnd := pos +1
44.                 r, t, n, h := h, None, val, None
45. return R
```

Algorithm to transform from (**P**, **S**, **D**) format to Scales

The Scale and its Context

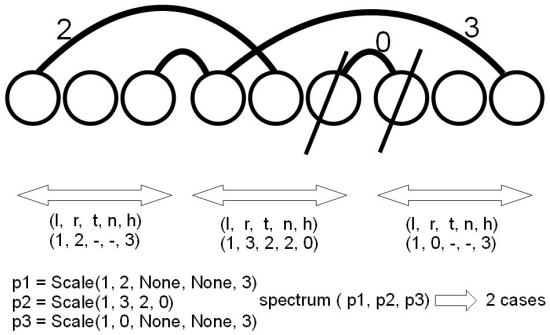
Now that we have the necessary structure for counting the number of cases, it is wanted to know how the kernel of function will be. All our functions are implemented around the algorithm presented in listing 2. That is, this algorithm will give a determined value for a length and three possible values for **r**, **t**, **n** and **h**.That is, **True**, **False** or **None** (both). From that, the result is the number of cases that ensures the query of the Scale. This function is an extension of function of Fibonacci, so that we have to implement it too. The different ways to implement Fibonacci function is out of the scope of this document.



The Algebra we need to work with the exposed algorithm to manage the Scale list is very similar to the delta of Dirac. That is, we will work with two opperations: the add is what insert cases for superposition and the product is what filters the queries into the wanted. The number of cases is proportional to the energy of the molecule the Scales are representing. So, in the way we are able of counting in the energy of tanglement we will be able of counting the different ways a molecule can meet properties.

Class Scale only takes the information of the scale using descriptors as in (6). Then we have until 16 possibilities of constructing determined cases: putting a True or a False to each four values in the tuple. That is the objective of method __call__. So the result is one of the 16 possible Contexts. Context is the class where we have to make that new algebra possible.

Adding and multiplying contexts, we will have the possibility of knowing the number of cases, filtering values, etc... And the algorithm which takes the list of Scales and convert them in the number of cases is called *spectrum*.



When the nature of a Scale is an extension of a list, the nature of the Context is an extension of a Table. So, every product of tables will work in only one product as an extension of inner join, outer join, etc..., and the add of tuples will include the number of cases it happens in the Context. So a Context is a set of tuples with its own cardinal. The sum of all cardinals is the cardinal of the Context.

$$
\begin{aligned}
Wa{\cdot}b{\cdot}c{\cdot}d \;=\; & Wa{\cdot}b{\cdot}0{\cdot}0 \;\&\; P0{\cdot}0{\cdot}0{\cdot}0 + Wa{\cdot}b{\cdot}0{\cdot}0 \;\&\; P0{\cdot}1{\cdot}c{\cdot}d + \\
& + Wa{\cdot}b{\cdot}0{\cdot}1 \;\&\; P1{\cdot}0{\cdot}0{\cdot}0 + Wa{\cdot}b{\cdot}0{\cdot}1 \;\&\; P1{\cdot}1{\cdot}c{\cdot}d + \\
& + Wa{\cdot}b{\cdot}1{\cdot}0 \;\&\; P0{\cdot}0{\cdot}0{\cdot}0 + Wa{\cdot}b{\cdot}1{\cdot}0 \;\&\; P0{\cdot}1{\cdot}c{\cdot}d + \\
& + Wa{\cdot}b{\cdot}1{\cdot}1 \;\&\; P1{\cdot}0{\cdot}0{\cdot}0 + Wa{\cdot}b{\cdot}1{\cdot}1 \;\&\; P1{\cdot}1{\cdot}c{\cdot}d
\end{aligned}
$$

You can add and multiply those cardinals if the cases are independent, and that is the objective of the Scale class: to offer independent Contexts to add and multiply. That is the reason we need to determine in the __call__ method the values of the Scale: those values are the bridges in the frontier of the Scale, where the cases are common.

Other example of use of Context

There exists other algorithm which uses Context class to calculate the Sharp of a well formed formula. We can get some ideas from **§2** to make the first step, that is: how we convert a boolean formula to the form described in (7).

$$
\begin{gathered}
\forall\, a_i, d_j \; Boolean\; Variables \wedge \forall\, b_i, c_i, \; Boolean\; Literals \wedge V_j \in \{1,0\} \\
\prod (a_i \Leftrightarrow b_i \wedge c_i) = 1 \\
\prod (d_j \Leftrightarrow V_j) = 1
\end{gathered}
\tag{7}
$$

To get the conversion, process starts generating the correspondent product after recognizing the operator we need to eliminate of the formula using lemmas in (8).

$$
\vdash (p \rightarrow q) \;\&\; p \rightarrow q \longrightarrow
\begin{array}{l}
\neg A = p \;\&\; \neg q \\
B = A \;\&\; p \\
\neg C = B \;\&\; \neg q \\
C = 1
\end{array}
$$

$$
\begin{aligned}
a:&\;\; f(A \wedge B, C) = 1 \Leftrightarrow f(Z, C) \wedge (Z \Leftrightarrow A \wedge B) = 1 \\
b:&\;\; f(A \vee B, C) = 1 \Leftrightarrow f(Z, C) \wedge (\neg Z \Leftrightarrow \neg A \wedge \neg B) = 1 \\
c:&\;\; f(A \Rightarrow B, C) = 1 \Leftrightarrow f(Z, C) \wedge (\neg Z \Leftrightarrow A \wedge \neg B) = 1
\end{aligned}
\tag{8}
$$

After using lemmas in (8), it is needed another conversion to eliminate the operator NOT on the left of the asignments. Something very easy of doing.

$$
\begin{array}{ll}
\neg A = p \;\&\; \neg q & D = p \;\&\; \neg q \\
B = A \;\&\; p \longrightarrow & B = \neg D \;\&\; p \\
\neg C = B \;\&\; \neg q & E = B \;\&\; \neg q \\
C = 1 & E = 0
\end{array}
$$

From that position, we can use the next rules in (9):

$$
\begin{gathered}
\delta X = \begin{Bmatrix} 1 & \leftarrow X\; negated \\ 0 & \leftarrow X\; asserted \end{Bmatrix} \\
A \Leftrightarrow B \wedge C \\
a: \#(A=0) = \#(B = \delta B) + \#(C = \delta C) - \#(B = \delta B, C = \delta C) \\
b: \#(A=1) = \#(B = 1 - \delta B, C = 1 - \delta C) \\
c: \forall\, H\; Boolean\; Literal: \#(H=0, H=1, ...) = 0
\end{gathered}
\tag{9}
$$

Those rules must be used with the algebra proposed with Context. That is, we have to work implicitly with the terms founded into the brackets like the tuples into the Context. Conceptually we can see the idea in (10).

$$
\begin{gathered}
\#(X = x) = \#(X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n) \\
\#(\beta, X = x, \alpha) = \#(\alpha, X = x, \beta) = \#(\beta, X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n, \alpha)
\end{gathered}
\tag{10}
$$

So, at last we only have to use the programming tecnic memoize to ensure we do not call twice the same variable to calculate its sharp.

D = p & ¬q	X = Y & ¬Z
B = ¬D & p	#(X=0)=#(Y=0)+#(Z=1)-#(Y=0,Z=1)
E = B & ¬q	#(X=1) = #(Y=1, Z=0)
E = 0	#(H=1, H=0, ...) = 0, ¥H

#(E=0) = #(B=0) + #(q=1)- #(B=0, q=1)= 3+2- #(p=0,q=1)=4
#(B=0) = #(p=0)+ #(D=1)- #(p=0, D=1)=2+1-0=3
#(D=1) = #(p=1, q=0)

Algorithm needed uses the Context class and is described in the last listing:

```
memo={}
def _sharp(rules, obj):
    global memo
    if not abs(obj) in rules.keys():
        return Context(1,Dic={abs(obj):int(obj>0)}),Context()
    if obj>0:
        if not rules[obj][0] in memo:
            memo[rules[obj][0]]=_sharp(rules,rules[obj][0])
        R1=memo[rules[obj][0]]
        if not rules[obj][1] in memo:
            memo[rules[obj][1]]=_sharp(rules,rules[obj][1])
        R2=memo[rules[obj][1]]
        return R1[0]*R2[0]+R1[1]*R2[1],R1[0]*R2[1]+R1[1]*R2[0]
    else:
        if not rules[-obj][0] in memo:
            memo[rules[-obj][0]]=_sharp(rules,-rules[-obj][0])
        R1 = memo[rules[-obj][0]]
        if not rules[-obj][1] in memo:
            memo[rules[-obj][1]]=_sharp(rules,-rules[-obj][1])
        R2 = memo[rules[-obj][1]]
        return R1[0]+R2[0]+R1[1]*R2[1],R1[1]+R2[1]+R1[0]*R2[0]

def sharp(rules, obj):
    S,R=_sharp(rules,obj)
    casos=0
    for atuple in S.tuples:
        casos+=2**len([X for X in atuple[1:] if X is None])
    for atuple in R.tuples:
        casos-=2**len([X for X in atuple[1:] if X is None])
    return casos

def test1():
    MP={4:(2,-3),5:(-4,2),6:(5,-3)}
    return sharp(MP,-6)
```

This algorithm works in P and we can easily use on a piece of paper manually. What it must be considered is the number of independent variables to calculate in the main function the exponent to add, because the only columns of every Context objects are the independent variables.

Conclussions

As has been shown, the Context class is not simply a new class. That class hides an algebra able of working with more power than the usual. So, what a good idea could be is to develope the use of this class. From the idea of table we got the SQL, though in that way because the computers of those days. Now we have a new oportunity to improve a new more conceptual standard which will give us ideas not imagined until now.

[1] J.M. Dato, Approaching to Hosoya Index, *International Journal Of Chemical And Pharmaceutical Research.* **2015**, Vol 1, No. 3.
[2] J.M. Dato, Relationship between P and NP. The ultimate definition, *American Open Computer Science Journal.* **2015**, Vol 2, No. *1*.

```python
class Context:
    def __init__(self,cardinal=0, columns=[0], tuples=[], Dic={}):
        'La columna 0 se reserva para apuntar los casos'
        if not Dic:
            self.cardinal=cardinal
            self.columns=columns if columns[0]==0 else [0]+columns
            self.tuples=tuples
        else:
            self.cardinal=Dic[None] if None in Dic.keys() else cardinal
            self.columns=[0]+[X for X in Dic.keys() if not X is None]
            self.tuples=[[self.cardinal]+[Dic[X] for X in Dic.keys()\
                                          if not X is None]]
    def __repr__(self):
        #self.m() #DEBUG MODE
        return "Context"+repr((self.cardinal,self.columns,self.tuples))
    def clone(self):
        return Context(self.cardinal,
                       self.columns[:],
                       [t[:] for t in self.tuples])
    def __bool__(self):
        return self.cardinal>0
    @staticmethod
    def comp(X,Y):
        'If Y is compatible con X'
        return X is None or Y is None or X==Y
    @staticmethod
    def AND(X,Y):
        if X is None:
            return Y
        elif Y is None or X==Y:
            return X
        else:
            return 2
    def __getuniqueitem(self, column):
        'called by __getitem__'
        P=self.columns.index(abs(column))
        S=0
        for atuple in self.tuples:
            if Context.comp(atuple[P],int(column>0)):
                S+=atuple[0]
        return S
```

```
def __getitem__(self, columns):
    if columns is None:
        return self.cardinal
    if type(columns) is int:
        return self.__getuniqueitem(columns)
    P=[self.columns.index(abs(c)) for c in columns]
    S=0
    for atuple in self.tuples:
        incompatible=False
        for i in range(len(columns)):
            if not Context.comp(atuple[P[i]],int(columns[i]>0)):
                incompatible=True
        if not incompatible:
            S+=atuple[0]
    return S
def _filter(self, columns, atuple):
    result=self.clone()
    for C in range(len(columns)):
        if not columns[C] in self.columns:
            result.columns.append(columns[C])
            result.tuples=[t+[atuple[C]] for t in result.tuples]

    result.cardinal=0
    for t in range(len(self.tuples)):
        for i in range(1,len(self.columns)):
            if self.columns[i] in columns:
                result.tuples[t][i]=Context.AND(
                    self.tuples[t][i],
                    atuple[columns.index(self.columns[i])])
                if result.tuples[t][i]==2:
                    result.tuples[t][0]=0
        result.tuples[t][0]*=atuple[0]
        result.cardinal+=result.tuples[t][0]

    result.tuples=[t for t in result.tuples if t[0]>0]
    return result
def __mul__(self, other):
    if not self or not other:
        return Context()
    R=Context()
    for atuple in other.tuples:
        R+=self._filter(other.columns,atuple)
    return R
```

```
def __add__(self,other):
    if not self:
        return other
    if not other:
        return self
    R=self.clone()
    for atuple in other.tuples:
        R=R.overlay(other.columns,atuple)
    return R
def insert(self,*atuple):
    'Must be incompatible with the rest'
    self.tuples.append(list(atuple))
    self.cardinal+=atuple[0]
@staticmethod
def cont(tX,tY):
    'If all cases of tY in tX'
    for j in range(len(tX)):
        if not (tX[j] is None or tX[j]==tY[j]):
            return False
    return True
@staticmethod
def adjust(col1,tup1,col2,tup2):
    'If all cases of tup2 in tup1'
    for p2 in range(1,len(col2)):
        if not col2[p2] in col1:
            return False
        p1=col1.index(col2[p2])
        if not (tup1[p1] is None or tup1[p1]==tup2[p2]):
            return False
    for p1 in range(1,len(col1)):
        if not col1[p1] in col2:
            if not tup1[p1] is None:
                return False
    return True
```

```
def overlay(self, columns, atuple):
    result=self.clone()
    for t in result.tuples:
        if Context.adjust(result.columns, t, columns, atuple):
            t[0]+=atuple[0]
            result.cardinal+=atuple[0]
            return result
    newTuple=[None]*len(result.columns)
    for p in range(len(columns)):
        if not columns[p] in result.columns:
            result.columns.append(columns[p])
            result.tuples=[t+[None] for t in result.tuples]
            newTuple.append(atuple[p])
        else:
            newTuple[result.columns.index(columns[p])]=atuple[p]
    result.insert(*newTuple)
    return result

def m(self):
    print(*self.columns,sep='\t')
    for t in self.tuples:
        print(*t,sep='\t')
    print(self.cardinal)
```